Europäisches Patentamt
European Patent Office
Office européen des brevets

Bescheinigung Certificate Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. Patent application No. Demande de brevet n°

02027277.9

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

R C van Dijk

Anmeldung Nr:
Application no.: 02027277.9
Demande no:

Anmeldetag:
Date of filing: 06.12.02
Date de dépôt:

Anmelder/Applicant(s)/Demandeur(s):

PACT XPP Technologies AG
Muthmannstrasse 1
80939 München
ALLEMAGNE

Bezeichnung der Erfindung/Title of the invention/Titre de l'invention:
(Falls die Bezeichnung der Erfindung nicht angegeben ist, siehe Beschreibung.
If no title is shown please refer to the description.
Si aucun titre n'est indiqué se referer à la description.)

A method for compiling high-level language programs to a reconfigurable data-flow processor

In Anspruch genommene Priorität(en) / Priority(ies) claimed /Priorité(s) revendiquée(s)
Staat/Tag/Aktenzeichen/State/Date/File no./Pays/Date/Numéro de dépôt:

Internationale Patentklassifikation/International Patent Classification/
Classification internationale des brevets:

G06F9/45

Am Anmeldetag benannte Vertragstaaten/Contracting states designated at date of filing/Etats contractants désignées lors du dépôt:

AT BE BG CH CY CZ DE DK EE ES FI FR GB GR IE IT LI LU MC NL PT SE SI SK

1 Introduction

This document describes a method for compiling a subset of a high-level programming language (HLL) like C or FORTRAN, extended by port access functions, to a reconfigurable data-flow processor (RDFP) as described in Section 3. The program is transformed to a configuration of the RDFP.

This method can be used as part of an extended compiler for a hybrid architecture consisting of standard host processor and a reconfigurable data-flow coprocessor. The extended compiler handles a full HLL like standard ANSI C. It maps suitable program parts like inner loops to the coprocessor and the rest of the program to the host processor. It is also possible to map separate program parts to separate configurations. However, these extensions are not subject of this document.

Whereas one aspect is claimed in particular, the skilled person will identify numerous independent inventions in this application which are believed to be patentable by themselves.

2 Compilation Flow

This section briefly describes the phases of the compilation method.

2.1 Frontend

The compiler uses a standard frontend which translates the input program (e. g. a C program) into an internal format consisting of an abstract syntax tree (AST) and symbol tables. The frontend also performs well-known compiler optimizations as constant propagation, dead code elimination, common subexpression elimination etc. For details, refer to any compiler construction textbook like [1]. The SUIF compiler [2] is an example of a compiler providing such a frontend.

2.2 Control/Dataflow Graph Generation

Next, the program is mapped to a control/dataflow graph (CDFG) consisting of connected RDFP functions. This phase is the main subject of this document and presented in Section 4.

2.3 Configuration Code Generation

Finally, the last phase directly translates the CDFG to configuration code used to program the RDFP. For PACT XPP™ Cores, the configuration code is generated as an NML (Native Mapping Language) file.

3 Configurable Objects and Functionality of a RDFP

This section describes the configurable objects and functionality of a RDFP. A possible implementation of the RDFP architecture is a PACT XPP™ Core. Here we only describe the minimum requirements for a RDFP for this compilation method to work. The only data types considered are multi-bit words called *data* and single-bit control signals called *events*. Data and events are always processed as *packets*, cf. Section 3.2. Event packets are called 1-events or 0-events, depending on their bit-value.

3.1 Configurable Objects and Functions

An RDFP consists of an array of configurable objects and a communication network. Each object can be configured to perform certain functions (listed below). It performs the same function repeatedly until the configuration is changed. The array needs not be completely uniform, i. e. not all objects need to be able to perform all functions. E. g., a RAM function can be implemented by a specialized RAM object which cannot perform any other functions. It is also possible to combine several objects to a "macro" to realize certain functions. Several RAM objects can, e. g., be combined to realize a RAM function with larger storage.

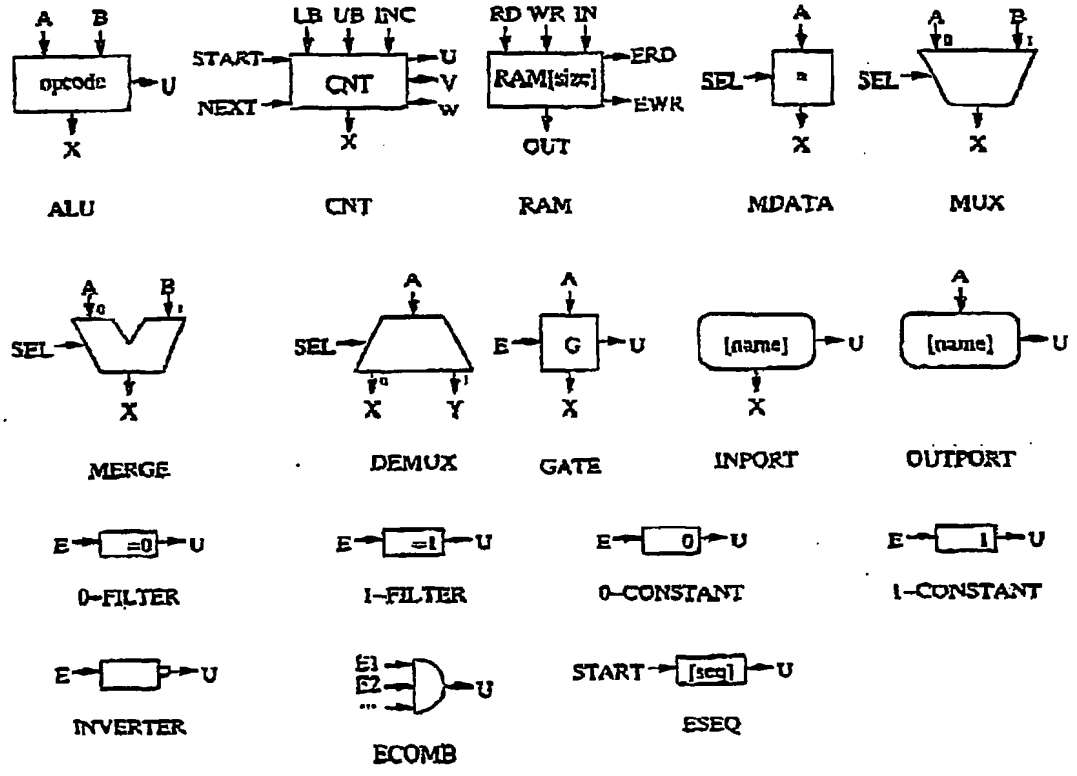


Figure 1: Functions of an RDFP

The following functions for processing data and event packets can be configured into an RDFP. See Fig. 1 for a graphical representation.

- ALU[opcode]: ALUs perform common arithmetical and logical operations on data. ALU functions ("opcodes") must be available for all operations used in the HLL.[1] ALU functions have two data inputs A and B, and one data output X. Comparators have an event output U instead of the data output. They produce a 1-event if the comparison is true, and a 0-event otherwise.

[1] Otherwise programs containing operations which do not have ALU opcodes in the RDFP must be excluded from the supported HLL subset or substituted by "macros" of existing functions.

- CNT: A counter function which has data inputs LB, UB and INC (lower bound, upper bound and increment) and data output X (counter value). A packet at event input START starts the counter, and event input NEXT causes the generation of the next output value (and output events) or causes the counter to terminate if UB is reached. If NEXT is not connected, the counter counts continuously. The output events U, V, and W have the following functionality: For a counter counting N times, N-1 0-events and one 1-event are generated at output U. At output V, N 0-events are generated, and at output W, N 0-events and one 1-event are created. The 1-event at W is only created after the counter has terminated, i. e. a NEXT event packet was received after the last data packet was output.

- RAM[size]: The RAM function stores a fixed number of data words ("size"). It has a data input RD and a data output OUT for reading at address RD. Event output ERD signals completion of the read access. For a write access, data inputs WR and IN (address and value) and data output OUT is used. Event output EWR signals completion of the write access. ERD and EWR always generate 0-events. Note that external RAM can be handled as RAM functions exactly like internal RAM.

- GATE: A GATE synchronizes a data packet at input A back and an event packet at input E. When both inputs have arrived, they are both consumed. The data packet is copied to output X, and the event packet to output U.

- MUX: A MUX function has 2 data inputs A and B, an event input SEL, and a data output X. If SEL receives a 0-event, input A is copied to output X and input B discarded. For a 1-event, B is copied and A discarded.

- MERGE: A MERGE function has 2 data inputs A and B, an event input SEL, and a data output X. If SEL receives a 0-event, input A is copied to output X, but input B is *nor* discarded. The packet is left at the input B instead. For a 1-event, B is copied and A left at the input.

- DEMUX: A DEMUX function has one data input A, an event input SEL, and two data outputs X and Y. If SEL receives a 0-event, input A is copied to output X, and no packet is created at output Y. For a 1-event, A is copied to Y, and no packet is created at output X.

- MDATA: A MDATA function multiplicates data packets. It has a data input A, an event input SEL, and a data output X. If SEL receives a 1-event, a data packet at A is consumed and copied to output X. For all subsequent 0-event at SEL, a copy of the input data packet is produced at the output without consuming new packets at A. Only if another 1-event arrives at SEL, the next data packet at A is consumed and copied.[2]

- INPORT[name]: Receives data packets from outside the RDFP through input port "name" and copies them to data output X. If a packet was received, a 0-event is produced at event output U, too. (Note that this function can only be configured at special objects connected to external busses.)

- OUTPORT[name]: Sends data packets received at data input A to the outside of the RDFP through output port "name". If a packet was sent, a 0-event is produced at event output U, too. (Note that this function can only be configured at special objects connected to external busses.)

Additionally, the following functions manipulate only event packets:

[2]Note that this can be implemented by a MERGE with special properties on XPP™.

- 0-FILTER, 1-FILTER: A FILTER has an input E and an output U. A 0-FILTER copies a 0-event from E to U, but 1-EVENTs at E are discarded. A 1-FILTER copies 1-events and discards 0-events.
- INVERTER: Copies all events from input E to output U but inverts its value.
- 0-CONSTANT, 1-CONSTANT: 0-CONSTANT copies all events from input E to output U, but changes them all to value 0. 1-CONSTANT changes all to value 1.
- ECOMB: Combines two or more inputs E1, E2, E3..., producing a packet at output U. The output is a 1-event if and only if one or more of the input packets are 1-events (logical *or*). A packet must be available at all inputs before an ouput packet is produced.[3]
- ESEQ[seq]: An ESEQ generates a sequence "seq" of events, e.g. "0001", at its output U. If it has an input START, one entire sequence is generated for each event packet arriving at U. The sequence is only repeated if the next event arrives at U. However, if START is not connected, ESEQ constantly repeats the sequence.

Note that ALU, MUX, DEMUX, GATE and ECOMB functions behave like their equivalents in classical dataflow machines [3, 4].

3.2 Packet-based Communication Network

The communication network of an RDFP can connect an outputs of one object (i.e. its respective function) to the input(s) of one or several other objects. This is usually achieved by busses and switches. By placing the functions properly on the objects, many functions can be connected arbitrarily up to a limit imposed by the device size. As mentioned above, all values are communicated as packets. A separate communication network exists for data and event packets. The packets synchronize the functions as in a dataflow machine with acknowledge [3]. I. e., the function only executes when all input packets are available (apart from the non-strict exceptions as described above). The function also stalls if the last output packet has not been consumed. Therefore a data-flow graph mapped to an RDFP self-synchronizes its execution without the need for external control. Only if two or more function outputs (data or event) are connected to the same function input ("N to 1 connection"), the self-synchronization is disabled.[4] The user has to ensure that only one packet arrives at a time in a correct CDFG. Otherwise a packet might get lost, and the value resulting from combining two or more packets is undefined. However, a function output can be connected to many function inputs ("1 to N connection") without problems.

There are some special cases:

- A function input can be *preloaded* with a distinct value during configuration. This packet is consumed like a normal packet coming from another object.
- A function input can be defined as *constant*. In this case, the packet at the input is reproduced repeatedly for each function execution.

[3] Note that this function is implemented by the EAND operator on the XPP™.

[4] Note that on XPP™ Cores, a "N to 1 connection" for events is realized by the EOR function, and for data by just assigning several outputs to an input.

An RDFP requires register delays in the dataflow. Otherwise very long combinational delays and asynchronous feedback is possible. We assume that delays are inserted at the inputs of some functions (like for most ALUs) and in some routing segments of the communication network. Note that registers change the timing, but not the functionality of a correct CDFG.

4 Configuration Generation

4.1 Language Definition

The following HLL features are not supported by the method described here:

- pointer operations
- library calls, operating system calls (including standard I/O functions)
- recursive function calls (Note that non-recursive function calls can be eliminated by function inlining and therefore are not considered here.)
- All scalar data types are converted to type integer. Integer values are equivalent to *data* packets in the RDFP. Arrays (possibly multi-dimensional) are the only composite data types considered.

The following additional features are supported:

INPORTS and OUTPORTS can be accessed by the HLL functions *getstream(name, value)* and *putstream(name, value)* respectively.

4.2 Mapping of High-Level Language Constructs

This method converts a HLL program to a CDFG consisting of the RDFP functions defined in Section 3.1. Before the processing starts, all HLL program arrays are mapped to RDFP RAM functions. An array x is mapped to RAM RAM(x). If several arrays are mapped to the same RAM, an offset is assigned, too. The RAMs are added to an initially empty CDFG. There must be enough RAMs of sufficient size for all program arrays.

The CDFG is generated by a traversal of the AST of the HLL program. It processes the program statement by statement and descends into the loops and conditional statements as appropriate. The following two pieces of information are updated at every program point[5] during the traversal:

- START points to an event output of a RDFP function. This output delivers a 0-event whenever the program execution reaches this program point. At the beginning, a 0-CONSTANT preloaded with an event input is added to the CDFG. (It delivers a 0-event immediately after configuration.) START initially points to its output. This event is used to start the overall program execution. The $START_{new}$ signal generated after a program part has finished executing is used as new START signal for the following program parts, or it signals termination of the entire program. The START

[5] In a program, *program points* are between two statements or before the beginning or after the end of a program component like a loop or a conditional statement.

events guarantee that the execution order of the original program is maintained wherever the data dependencies alone are not sufficient. This scheduling scheme is similar to a *one-hot controller* for digital hardware.

- VARLIST is a list of {*variable, function-output*} pairs. The pairs map integer variables or array elements to a CDFG function's output. The first pair for a variable in VARLIST contains the output of the function which produces the value of this variable valid at the current program point. New pairs are always added to the front of VARLIST. The expression VARDEF(var) refers to the *function-output* of the first pair with *variable* var in VARLIST.[6]

The following subsections systematically list all HLL program components and describe how they are processed, thereby altering the CDFG, START and VARLIST.

4.2.1 Integer Expressions and Assignments

Straight-line code without array accesses can be directly mapped to a data-flow graph. One ALU is allocated for each operator in the program. Because of the self-synchronization of the ALUs, no explicit control or scheduling is needed. Therefore processing these assignments does not access or alter START. The data dependences (as they would be exposed in the DAG representation of the program [1]) are analyzed through the processing of VARLIST. These assignments synchronize themselves through the data-flow. The data-driven execution automatically exploits the available instruction level parallelism.

All assignments evaluate the right-hand side (RHS) or source expression. This evaluation results in a pointer to a CDFG object's output (or pseudo-object as defined below). For integer assignments, the left-hand side (LHS) variable or destination is combined with the RHS result object to form a new pair {LHS, result(RHS)} which is added to the front of VARLIST.

The simplest statement is a constant assigned to an integer:[7]

```
a = 5;
```

It doesn't change the CDFG, but adds {a, 5} to the front of VARLIST. The constant 5 is a "pseudo-object" which only holds the value, but does not refer to a CDFG object. Now VARDEF(a) equals 5 at subseqent program points before a is redefined.

Integer assignments can also combine variables already defined and constants:

```
b = a * 2 + 3;
```

In the AST, the RHS is already converted to an expression tree. This tree is transformed to a combination of old and new CDFG objects (which are added to the CDFG) as follows: Each operator (internal node) of the tree is substituted by an ALU with the opcode corresponding to the operator in the tree. If a leaf node is a constant, the ALU's input is directly connected to that constant. If a leaf note is an integer variable var, it is looked up in VARLIST, i. e. VARDEF(var) is retrieved. Then VARDEF(var) (an output of an already existing object in CDFG or a constant) is connected to the ALU's input. The output of the ALU corresponding to the root operator in the expression tree is defined as the *result* of the RHS. Finally, a new pair {LHS, result(RHS)} is added to VARLIST. If the two assignments above are processed, the

[6] This method of using a VARLIST is adapted from the Transmogrifier C compiler [5].

[7] Note that we use C syntax for the following examples.

CDFG with two ALUs in Fig. 2 is created.[8] Outputs occurring in VARLIST are labeled by Roman numbers. After these two assignments, VARLIST = [{b, I}, {a, 5}]. (The front of the list is on the left side.) Note that all inputs connected to a constant (whether direct from the expression tree or retrieved from VARLIST) must be defined as constant. Inputs defined as constants have a small c next to the input arrow in Fig. 2.

4.2.2 Conditional Integer Assignments

For conditional if-then-else statements containing only integer assignments, objects for condition evaluation are created first. The object event output indicating the condition result is kept for choosing the correct branch result later. Next, both branches are processed in parallel, using separate copies VARLIST1 and VARLIST2 of VARLIST. (VARLIST itself is not changed.) Finally, for all variables added to VARLIST1 *or* VARLIST2, a new entry for VARLIST is created (combination phase). The valid definitions from VARLIST1 and VARLIST2 are combined with a MUX function, and the correct input is selected by the condition result. For variables only defined in one of the two branches, the multiplexer uses the result retrieved from the original VARLIST for the other branch. If the original VARLIST does not have an entry for this variable, a special "undefined" constant value is used. However, in a functionally correct program this value will never be used. As an optimization, only variables *live* [1] after the if-then-else structure need to be added to VARLIST in the combination phase.[9]

Consider the following example:

```
i = 7;
a = 3;
if (i < 10) {
  a = 5;
  c = 7;
}
else {
  c = a - 1;
  d = 0;
}
```

Fig. 3 shows the resulting CDFG. Before the if-then-else construct, VARLIST = [{a, 3}, {i, 7}]. After processing the branches, for the then branch, VARLIST1 = [{c, 7}, {a, 5}, {a, 3}, {i, 7}]. and for the else branch, VARLIST2 = [{d, 0}, {c, I}, {a, 3}, {i, 7}]. After combination, VARLIST = [{d, II}, {c, III}, {a, IV}, {a, 3}, {i, 7}].

Note that case- or switch-statements can be processed, too, since they can – without loss of generality – be converted to nested if-then-else statements.

Processing conditional statements this way does not require explicit control and does not change START. Both branches are executed in parallel and synchronized by the data-flow. It is possible to pipeline the dataflow for optimal throughput.

[8] Note that the input and output names can be deduced from their position, cf. Fig. 1. Also note that the compiler front-end would normally have substituted the second assignment by `b = 13` (constant propagation). For the simplicity of this explanation, no frontend optimizations are considered in this and the following examples.

[9] Definition: A variable is *live* at a program point if its value is read at a statement reachable from here without intermediate redefinition.

4.2.3 General Conditional Statements

Conditional statements containing either array accesses (cf. Section 4.2.7 below) or inner loops cannot be processed as described in Section 4.2.2. Data packets must only be sent to the active branch. This is achieved by the implementation shown in Fig. 8, similar to the method presented in [4].

A dataflow analysis is performed to compute *used sets* `use` and *defined sets* `def` [1] of both branches.[10] For the current VARLIST entries of all variables in $IN = use(thenbody) \cup def(thenbody) \cup use(elsebody) \cup def(elsebody) \cup use(header)$, DEMUX functions controlled by the IF condition are inserted. Note that arrows with double lines in Fig. 8 denote connections for all variables in IN, and the shaded DEMUX function stands for several DEMUX functions, one for each variable in IN. The DEMUX functions forward data packets only to the selected branch. New lists VARLIST1 and VARLIST2 are compiled with the respective outputs of these DEMUX functions. The then-branch is processed with VARLIST1, and the else branch with VARLIST2. Finally, the output values are combined. OUT contains the new values for the same variables as in IN. Since only one branch is ever activated there will not be a conflict due to two packets arriving simultaneously. The combinations will be added to VARLIST after the conditional statement. If the IF execution shall be pipelined, MERGE opcodes for the output must be inserted, too. They are controlled by the condition like the DEMUX functions.

The following extension with respect to [4] is added (dotted lines in Fig. 8) in order to control the execution as mentioned above with START events: The START input is ECOMB-combined with the condition output and connected to the SEL input of the DEMUX functions. The START inputs of thenbody and elsebody are generated from the ECOMB output sent through a 1-FILTER and a 0-CONSTANT[11] or through a 0-FILTER, respectively. The overall $START_{new}$ output is generated by a simple "2 to 1 connection" of thenbody's and elsebody's $START_{new}$ outputs. With this extension, arbitrarily nested conditional statements or loops can be handled within thenbody and elsebody.

4.2.4 WHILE Loops

WHILE loops are processed similarly to the scheme presented in [4], cf. Fig. 9. As in Section 4.2.3, double line connections and shaded MERGE and DEMUX functions represent duplication for all variables in IN. Here $IN = use(whilebody) \cup def(whilebody) \cup use(header)$. The WHILE loop executes as follws: In the first loop iteration, the MERGE functions select all input values from VARLIST at loop entry (SEL=0). The MERGE outputs are connected to the header and the DEMUX functions. If the while condition is true (SEL=1), the input values are forwarded to the whilebody, otherwise to OUT. The output values of the while body are fed back to whilebody's input via the MERGE and DEMUX operators as long as the condition is true. Finally, after the last iteration, they are forwarded to OUT. The outputs are added to the new VARLIST.[12]

Two extensions with respect to [4] are added (dotted lines in Fig. 9):

[10] A variable is *used* in a statement (and hence in a program region containing this statement) if its value is read. A variable is *defined* in a statement (or region) if a new value is assigned to it.

[11] The 0-CONSTANT is required since START events must always be 0-events.

[12] Note that the MERGE function for variables not live at the loop's beginning and the whilebody's beginning can be removed since its output is not used. For these variables, only the DEMUX function to output the final value is required. Also note that the MERGE functions can be replaced by simple "2 to 1 connections" if the configuration process guarantees that packets from IN1 always arrive at the DEMUX's input before feedback values arrive.

- In [4], the SEL input of the MERGE functions is preloaded with 0. Hence the loop execution begins immediately and can be executed only once. Instead, we connect the START input to the MERGE's SEL input ("2 to 1 connection" with the header output). This allows to control the time of the start of the loop execution and to restart it.
- The whilebody's START input is connected to the header output, sent through a 1-FILTER/0-CONSTANT combination as above (generates a 0-event for each loop iteration). By ECOMB-combining whilebody's $START_{new}$ output with the header output for the MERGE functions' SEL inputs, the next loop iteration is only started after the previous one has finished. The while loop's $START_{new}$ output is generated by filtering the header output for a 0-event.

With these extensions, arbitrarily nested conditional statements or loops can be handled within whilebody.

4.2.5 FOR Loops

FOR loops are particularly regular WHILE loops. Therefore we could handle them as explained above. However, our RDFP features the special counter function CNT and the data packer multiplication function MDATA which can be used for a more efficient implementation of FOR loops. This new FOR loop scheme is shown in Fig. 10.

A FOR loop is controlled by a counter CNT. The lower bound (LB), upper bound (UB), and increment (INC) expressions are evaluated like any other expressions (see Sections 4.2.1 and 4.2.7) and connected to the respective inputs.

As opposed to WHILE loops, a MERGE/DEMUX combination is only required for variables in $IN1 = def(forbody)$, i. e. those *defined* in forbody.[13] IN1 does not contain variables which are only *used* in forbody, LB, UB, or INC, and does also not contain the loop index variable. Variables in IN1 are processed as in WHILE loops, but the MERGE and DEMUX functions' SEL input is connected to CNT's W output. (The W output does the inverse of a WHILE loop's header output; it outputs a 1-event after the counter has terminated. Therefore the inputs of the MERGE functions and the outputs of the DEMUX functions are swapped here, and the MERGE functions' SEL inputs are preloaded with 1-events.)

CNT's X output provides the current value of the loop index variable. If the final index value is required (live) after the FOR loop, it is selected with a DEMUX function controlled by CNT's U event output (which produces one event for every loop iteration).

Variables in $IN2 = use(forbody) \setminus def(forbody)$, i. e. those defined outside the loop and only used (but not redefined) inside the loop are handled differently. Unless it is a constant value, the variable's input value (from VARLIST) must be reproduced in each loop iteration since it is consumed in each iteration. Otherwise the loop would stall from the second iteration onwards. The packers are reproduced by MDATA functions, with the SEL inputs connected to CNT's U output. The SEL inputs must be preloaded with a 1-event to select the first input. The 1-event provided by the last iteration selects a new value for the next execution of the entire loop.

[13] Note that the MERGE functions can be replaced by simple "2 to 1 connections" as for WHILE loops if the configuration process guarantees that packets from IN1 always arrive at the DEMUX's input before feedback values arrive.

The following control events (dotted lines in Fig. 10) are similar to the WHILE loop extensions, but simpler. CNT's START input is connected to the loop's overall START signal. $START_{new}$ is generated from CNT's W output, sent through a 1-FILTER and 0-CONSTANT. CNT's V output produces one 0-event for each loop iteration and is therefore used as forbody's START. Finally, CNT's NEXT input is connected to forbody's $START_{new}$ output.

For pipelined loops (as defined below in Section 4.2.6), loop iterations are allowed to overlap. Therefore CNT's NEXT input needs not be connected. Now the counter produces index variable values and control events as fast as they can be consumed. However, in this case CNT's W output in not sufficient as overall $START_{new}$ output since the counter terminates before the last iteration's forbody finishes. Instead, $START_{new}$ is generated from CNT's U output ECOMB-combined with forbody's $START_{new}$ output, sent through a 1-FILTER/0-CONSTANT combination. The ECOMB produces an event after termination of each loop iteration, but only the *last* event is a 1-event because only the last output of CNT's U output is a 1-event. Hence this event indicates that the last iteration has finished. Cf. Section 4.3 for a FOR loop example compilation with and without pipelining.

As for WHILE loops, these methods allow to process arbitrarily nested loops and conditional statements. The following advantages over WHILE loop implementations are achieved:

- One index variable value is generated by the CNT function each clock cycle. This is faster and smaller than the WHILE loop implementation which allocates a MERGE/DEMUX/ADD loop and a comparator for the counter functionality.
- Variables in IN2 (only used in forbody) are reproduced in the special MDATA functions and need not go through a MERGE/DEMUX loop. This is again faster and smaller than the WHILE loop implementation.

4.2.6 Vectorization and Pipelining

The method described so far generates CDFGs performing the HLL program's functionality on an RDFP. However, the program execution is unduly sequentialized by the START signals. In some cases, innermost loops can be *vectorized*. This means that loop iterations can overlap, leading to a pipelined dataflow through the operators of the loop body. The *Pipeline Vectorization* technique [6] can be easily applied to the compilation method presented here. As mentioned above, for FOR loops, the CNT's NEXT input is removed so that CNT counts continuously, thereby overlapping the loop iterations.

All loops without array accesses can be pipelined since the dataflow automatically synchronizes *loop-carried dependences*, i. e. dependences between a statement in one iteration and another statement in a subsequent iteration. Loops with array accesses can be pipelined if the array (i. e. RAM) accesses do not cause loop-carried dependences or can be transformed to such a form. In this case no RAM address is written in one and read in a subsequent iteration. Therefore the read and write accesses to the same RAM may overlap. This degree of freedom is exploited in the RAM access technique described below. Especially for dual-ported RAM it leads to considerable performance improvements.

4.2.7 Array Accesses

In contrast to scalar variables, array accesses have to be controlled explicitly in order to maintain the program's correct execution order. As opposed to normal dataflow machine models [3], a RDFP does

not have a single address space. Instead, the arrays are allocated to several RAMs. This leads to a different approach to handling RAM accesses and opens up new opportunities for optimization.

To reduce the complexity of the compilation process, array accesses are processed in two phases. Phase 1 uses "pseudo-functions" for RAM read and write accesses. A RAM read function has a RD data input (read address) and an OUT data output (read value), and a RAM write function has WR and IN data inputs (write address and write value). Both functions are labeled with the array the access refers to, and both have a START event input and a U event output. The events control the access order. In Phase 2 all accesses to the same RAM are combined and substituted by a single RAM function as shown in Fig. 1. This involves manipulating the data and event inputs and outputs such that the correct execution order is maintained and the outputs are forwarded to the correct part of the CDFG.

Phase 1 Since arrays are allocated to several RAMs, only accesses to the same RAM have to be synchronized. Accesses to different RAMs can occur concurrently or even out of order. In case of data dependencies, the accesses self-synchronize automatically. Within pipelined loops, not even read and write accesses to the same RAM have to be synchronized. This is achieved by maintaining separate START signals for every RAM or even separate START signals for RAM read and RAM write accesses in pipelined loops. At the end of a basic block [1][14], all $START_{new}$ outputs must be combined by a ECOMB to provide a START signal for the next basic block which guarantees that all array accesses in the previous basic block are completed. For pipelined loops, this condition can even be relaxed. Only after the loop exit all accesses have to be completed. The individual loop iterations need not be synchronized.

First the RAM addresses are computed. The compiler frontend's standard transformation for array accesses can be used, and a CDFG function's output is generated which provides the address. If applicable, the offset with respect to the RDFP RAM (as determined in the initial mapping phase) must be added. This output is connected to the pseudo RAM read's RD input (for a read access) or to the pseudo RAM write's WR input (for a write access). Additionally, the OUT output (read) or IN input (write) is connected. The START input is connected to the variable's START signal, and the U output is used as $START_{new}$ for the next access.

To avoid redundant read accesses, RAM reads are also registered in VARLIST. Instead of an integer variable, an array element is used as first element of the pair. However, a change in a variable occurring in an array index invalidates the information in VARLIST. It must then be removed from it.

The following example with two read accesses compiles to the intermediate CDFG shown in Fig. 12. The START signals refer only to variable a. STOP1 is the event connection which synchronizes the accesses. Inputs START (old), i and j should be substituted by the actual outputs resulting from the program before the array reads.

```
x = a[i];
y = a[j];
z = x + y;
```

Fig. 13 shows the translation of the following write access:

```
a[i] = x;
```

[14] A *basic block* is a program part with a single entry and a single exit point, i. e. a piece of straight-line code.

Phase 2 We now merge the pseudo-functions of all accesses to the same RAM and substitute them by a single RAM function. For all data inputs (RD for read access and WR and IN for write access), GATEs are inserted between the input and the RAM function. Their E inputs are connected to the respective START inputs of the original pseudo-functions. If a RAM is read and written at only one program point, the U output of the read and write access is moved to the ERD or EWR output, respectively. For example, the single access `a[i] = x;` from Fig. 13 is transformed to the final CDFG shown in Fig. 5.

However, if several read or several write accesses (i.e. pseudo-functions from different program points) to the same RAM occur, the ERD or EWR events are not specific anymore. But a $START_{new}$ event of the original pseudo function should only be generated for the respective program point, i.e. for the *current access*. This is achieved by connecting the START signals of all *other* accesses (pseudo-functions) of the same type (read or write) with the *inverted* START signal of the current access. The resulting signal produces an event for every access, but only for the current access a 1-event. This event is ECOMB-combined with the RAM's ERD or EWR output. The ECOMB's output will only occur after the access is completed. Because ECOMB OR-combines its event packets, only the current access produces a 1-event. Next, this event is filtered with a 1-FILTER and changed by a 0-CONSTANT, resulting in a $START_{new}$ signal which produces a 0-event only after the current access is completed as required.

For several accesses, several sources are connected to the RD, WR and IN inputs of a RAM. This disables the self-synchronization. However, since only one access occurs at a time, the GATEs only allow one data packet to arrive at the inputs.

For read accesses, the packets at the OUT output face the same problem as the ERD event packets: They occur for every read access, but must only be used (and forwarded to subsequent operators) for the current access. This can be achieved by connecting the OUT output via a DEMUX function. The Y output of the DEMUX is used, and the X output is left unconnected. Then it acts as a selective gate which only forwards packets if its SEL input receives a 1-event, and discards its data input if SEL receives a 0-event. The signal created by the ECOMB described above for the $START_{new}$ signal creates a 1-event for the current access, and a 0-event otherwise. Using it as the SEL input achieves exactly the desired functionality.

Fig. 4 shows the resulting CDFG for the first example above (two read accesses), after applying the transformations of Phase 2 to Fig. 12. STOP1 is now generated as follws: START(old) is inverted, "2 to 1 connected" to STOP1 (because it is the START input of the second read pseudo-function), ECOMB-combined with RAM's ERD output and sent through the 1-FILTER/0-CONSTANT combination. START(new)is generated similarly, but here START(old) is directly used and STOP1 inverted. The GATEs for input IN (i and j) are connected to START(old) and STOP1, respectively, and the DEMUX functions for outputs x and y are connected to the ECOMB outputs related to STOP1 and START(new).

Multiple write accesses use the same control events, but instead of one GATE per access for the RD inputs, one GATE for WR and one gate for IN (with the same E input) are used. The EWR output is processed like the ERD output for read accesses.

This transformation ensures that all RAM accesses are executed correctly, but it is not very fast since read or write accesses to the same RAM are not pipelined. The next access only starts after the previous one is completed, even if the RAM being used has several pipeline stages. This inefficiency can be removed as follws:

First continuous *sequences* of either read accesses or write accesses (not mixed) within a basic block are detected by checking for pseudo-functions whose U output is directly connected to the START input of another pseudo-function of the same RAM and the same type (read or write). For these sequences, it is

possible to stream data into the RAM rather than waiting for the previous access to complete. For this purpose, a combination of MERGE functions selects the RD or WR and IN inputs in the order given by the sequence. The MERGEs must be controlled by iterative ESEQs guaranteeing that the inputs are only forwarded in the desired order. Then only the first access in the sequence needs to be controlled by a GATE or GATEs. Similarly, the OUT outputs of a read access can be distributed more efficiently for a sequence. A combination of DEMUX functions with the same ESEQ control can be used. It is most efficient to arrange the MERGE and DEMUX functions as balanced binary trees.

The $START_{new}$ signal is generated as follows: For a sequence of length n, the START signal of the entire sequence is replicated n times by an ESEQ[00..1] function with the START input connected to the sequence's START. Its output is directly "N to 1 connected" with the other accesses' START signal (for single accesses) or ESEQ outputs sent through 0-CONSTANT (for access sequences), ECOMB-connected to EWR or ERD, respectively, and sent through a 1-FILTER/0-CONSTANT combination, similar to the basic method described above. Since only the last ESEQ output is a 1-event, only the *last* RAM access generates a $START_{new}$ as required. Alternatively, for read accesses, the generation of the last output can be sent through a GATE (without the E input connected), thereby producing a $START_{new}$ event.

Fig. 14 shows the optimized version of the first example (Figures 12 and 4) using the ESEQ-method for generating $START_{new}$, and Fig. 6 shows the final CDFG of the following, larger example with three array reads. Here the latter method for producing the $START_{new}$ event is used.

```
x = a[i];
y = a[j];
z = a[k];
```

If several read sequences or read sequences and single read accesses occur for the same RAM, 1-events for detecting the *current accesses* must be generated for sequences of read accesses. They are needed to separate the OUT-values relating to separate sequences. The ESEQ output just defined, sent through a 1-CONSTANT, achieves this. It is again "N to 1 connected" to the other accesses' START signals (for single accesses) or ESEQ outputs sent through 0-CONSTANT (for access sequences). The resulting event is used to control a first-stage DEMUX which is inserted to select the relevant OUT output data packets of the sequence as described above for the basic method. Refer to the second example (Figures 15 and 16) in Section 4.3 for a complete example.

4.2.8 Input and Output Ports

Input and output ports are processed similar to vector accesses. A read from an input port is like an array read without an address. The input data packet is sent to DEMUX functions which send it to the correct subsequent operators. The STOP signal is generated in the same way as described above for RAM accesses by combining the INPORT's U output with the current and other START signals.

Output ports control the data packets by GATEs like array write accesses. The STOP signal is also created as for RAM accesses.

4.3 More Examples

Fig. 7 shows the generated CDFG for the following for loop.

```
a = b + c;
for (i=0; i<=10; i++) {
  a = a + i;
  x[i] = k;
}
```

In this example, $IN1 = \{a\}$ and $IN2 = \{k\}$ (cf. Fig. 10). The MERGE function for variable a is replaced by a 2:1 data connection as mentioned in the footnote of Section 4.2.5. Note that only one data packet arrives for variables b, c and k, and one final packet is produced for a (out). forbody does not use a START event since both operations (the adder and the RAM write) are dataflow-controlled by the counter anyway. But the RAM's EWR output is the forbody's $START_{new}$ and connected to CNT's NEXT input. Note that the pipelining optimization, cf. Section 4.2.6, was not applied here. If it is applied (which is possible for this loop), CNT's NEXT input is not connected, cf. Fig. 11. Here, the loop iterations overlap. $START_{new}$ is generated from CNT's U output and forbody's $START_{new}$ (i.e. RAM's EWR output), as defined at the end of Section 4.2.5.

The following program contains a vectorizable (pipelined) loop with one write access to array (RAM) x and a sequence of two read accesses to array (RAM) y. After the loop, another single read access to y occurs.

```
z = 0;
for (i=0; i<=10; i++) {
  x[i] = i;
  z = z + y[i] + y[2*i];
}
a = y[k];
```

Fig. 15 shows the intermediate CDFG generated before the array access Phase 2 transformation is applied. The pipelined loop is controlled as follows: Within the loop, separate START signals for write accesses to x and read accesses to y are used. The reentry to the forbody is also controlled by two independent signals ("cycle1" and "cycle2"). For the read accesses, "cycle2" guarantees that the read y accesses occur in the correct order. But the beginning of an iteration for read y and write x accesses is not synchronized. Only at loop exit all accesses must be finished, which is guaranteed by signal "loop finished". The single read access is completely independent of the loop.

Fig. 16 shows the final CDFG after Phase 2. Note that "cycle1" is removed since a single write access needs no additional control, and "cycle2" is removed since the inserted MERGE and DEMUX functions automatically guarantee the correct execution order. The read y accesses are not independent anymore since they all refer to the same RAM, and the functions have been merged. ESEQs have been allocated to control the MERGE and DEMUX functions of the read sequence, and for the first-stage DEMUX functions which separate the read OUT values for the read sequence and for the final single read access. The ECOMBs, 1-FILTERs, 0-CONSTANTs and 1-CONSTANTs are allocated as described in Section 4.2.7. Phase 2, to generate correct control events for the GATEs and DEMUX functions.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] The Stanford SUIF Compiler Group. Homepage `http://suif.stanford.edu`.

[3] A. H. Veen. Dataflow architecture. *ACM Computing Surveys*, 18(4), December 1986.

[4] S. J. Allan and A. E. Oldehoeft. A flow analysis procedure for the translation of high-level languages to a data flow language. *IEEE Transactions on Computers*, C-29(9):826–831, September 1980.

[5] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *Proc. FPGAs for Custom Computing Machines*, pages 136–144. IEEE Computer Society Press, 1995.

[6] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2), February 2001.

Akte: PACT27a/

European patent application

Applicant: PACT XPP Technologies AG
Muthmannstraße 1
80939 München

Representative: European Patent Attorney
Claus Peter Pietruk
Heinrich-Lilienfein-Weg 5
D-76299 Karlsruhe
0 085 850

Title: A method for compiling high-level language programs to a reconfigurable data-flow processor

Claims

1. A method for providing configurations for a multidimensional array of coarse-grained and/or fine-grained arithmetic and/or logic cells according to a high-level-language comprising FOR-Loops,
wherein
a counter (4.2.5) is implemented in said array when a loop is to be configured into said array.

Fig. 1:

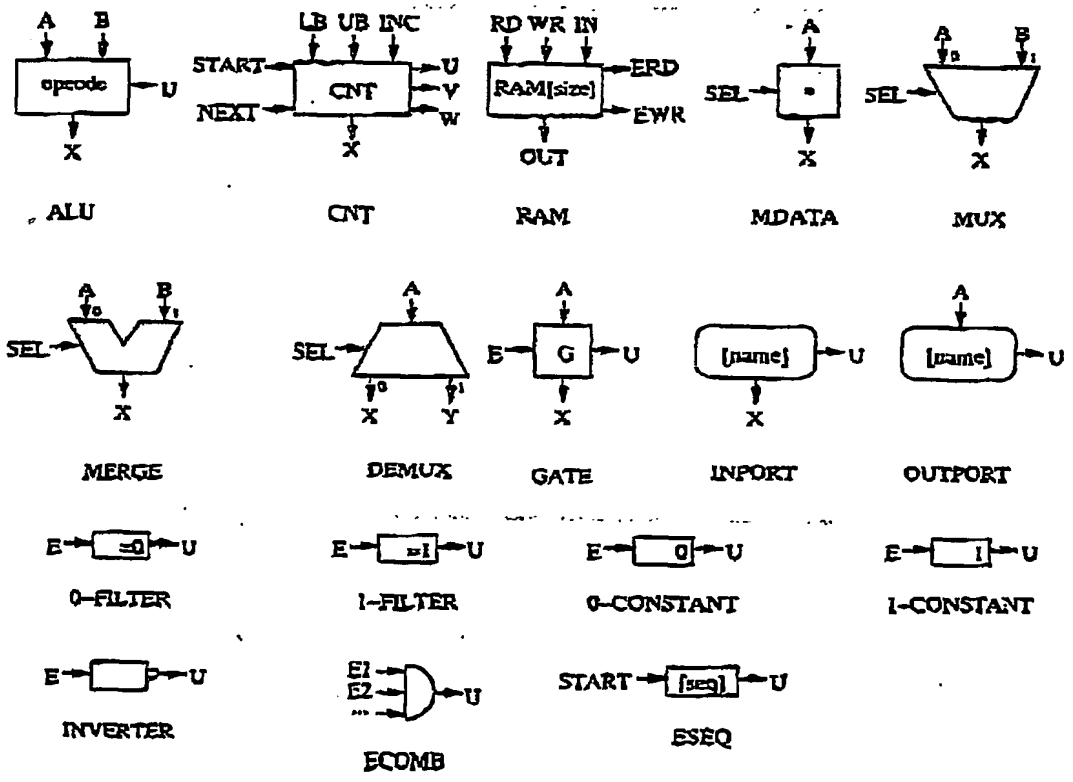

A
B
opcode
U
X
ALU
LB UB INC
START
NEXT
CNT
U
V
W
X
CNT
RD WR IN
RAM[size]
ERD
EWR
OUT
RAM
A
SEL
*
X
MDATA
A
B
SEL
X
MUX
A
B
SEL
X
MERGE
A
SEL
X
Y
DEMUX
A
E
G
U
X
GATE
[name]
U
X
INPORT
A
[name]
U
OUTPORT
E
=0
U
0-FILTER
E
=1
U
1-FILTER
E
0
U
0-CONSTANT
E
1
U
1-CONSTANT
E
U
INVERTER
E1
E2
U
ECOMB
START
[seq]
U
ESEQ

Fig. 2:

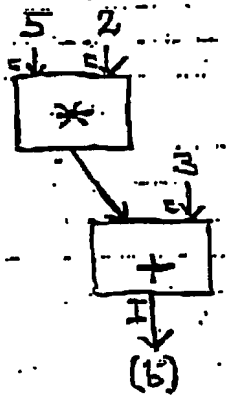


Fig. 3:

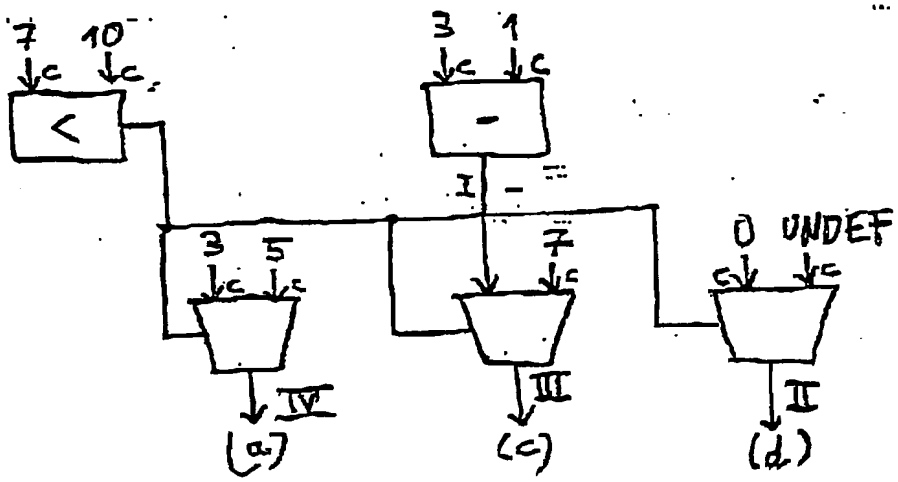


Fig. 4: [NOTE: One array was forgotten in this figure in original submission!]

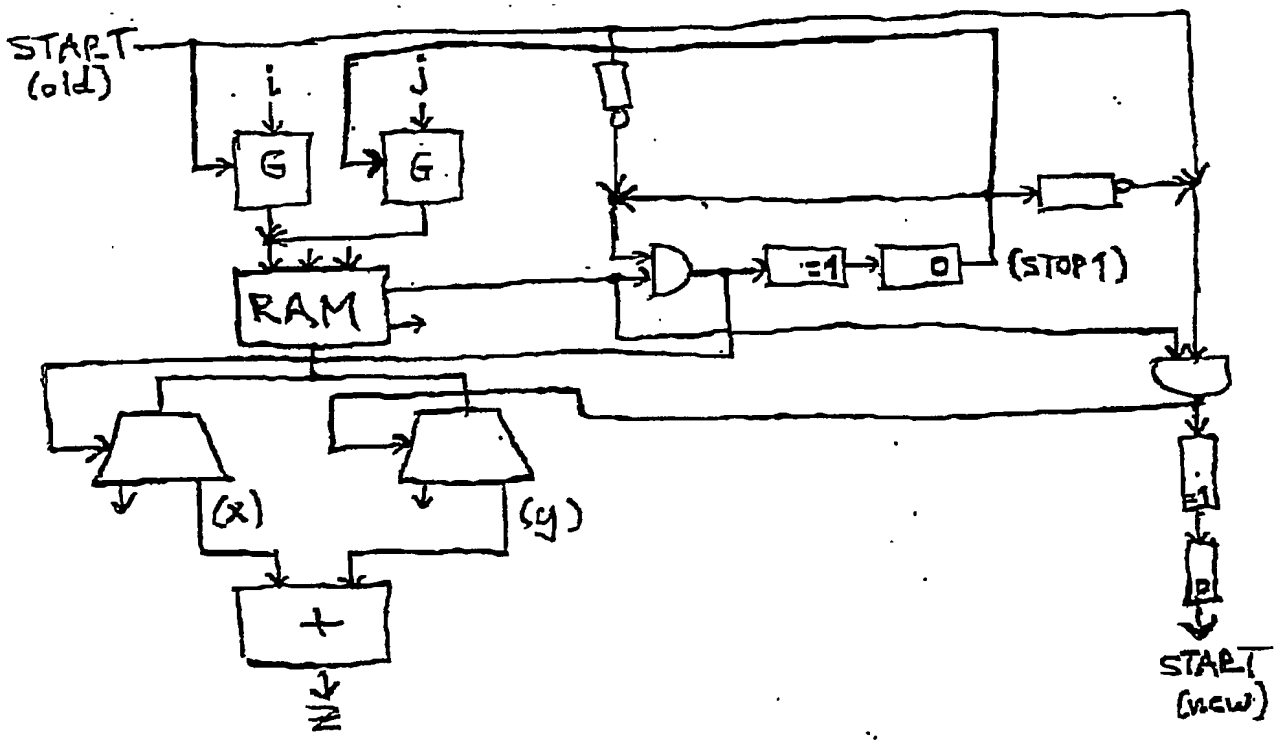


Fig. 5:

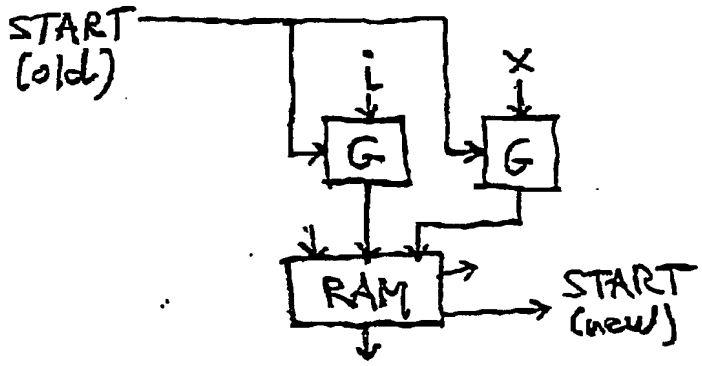

START
(old)
i
x
G
G
RAM
START
(new)

Fig. 6:

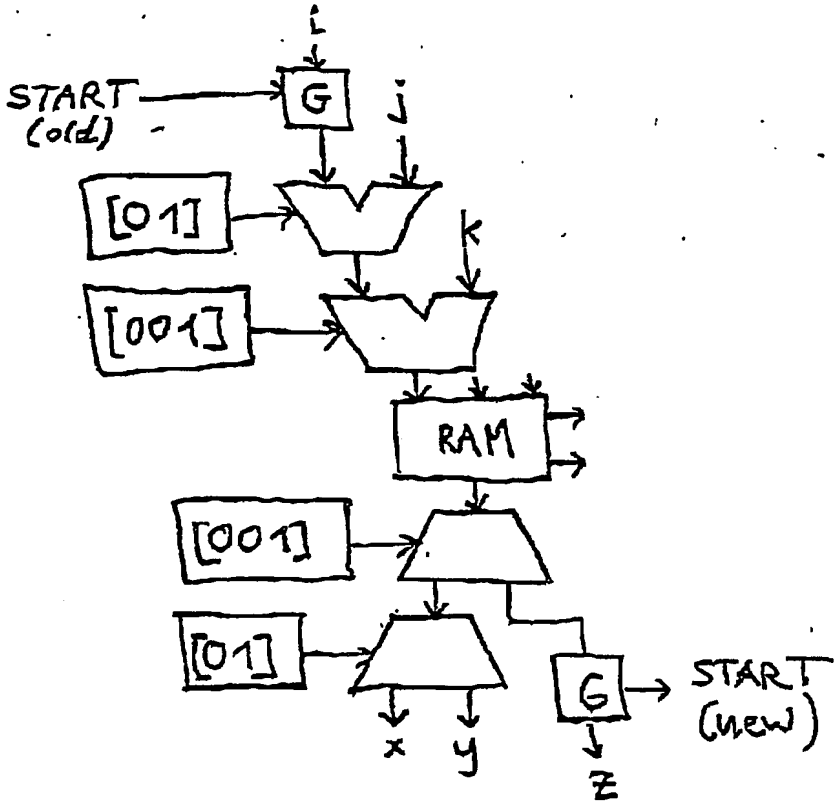

i
START
(old)
G
j
[01]
k
[001]
RAM
[001]
[01]
G
START
(new)
x
y
z

Fig. 7:

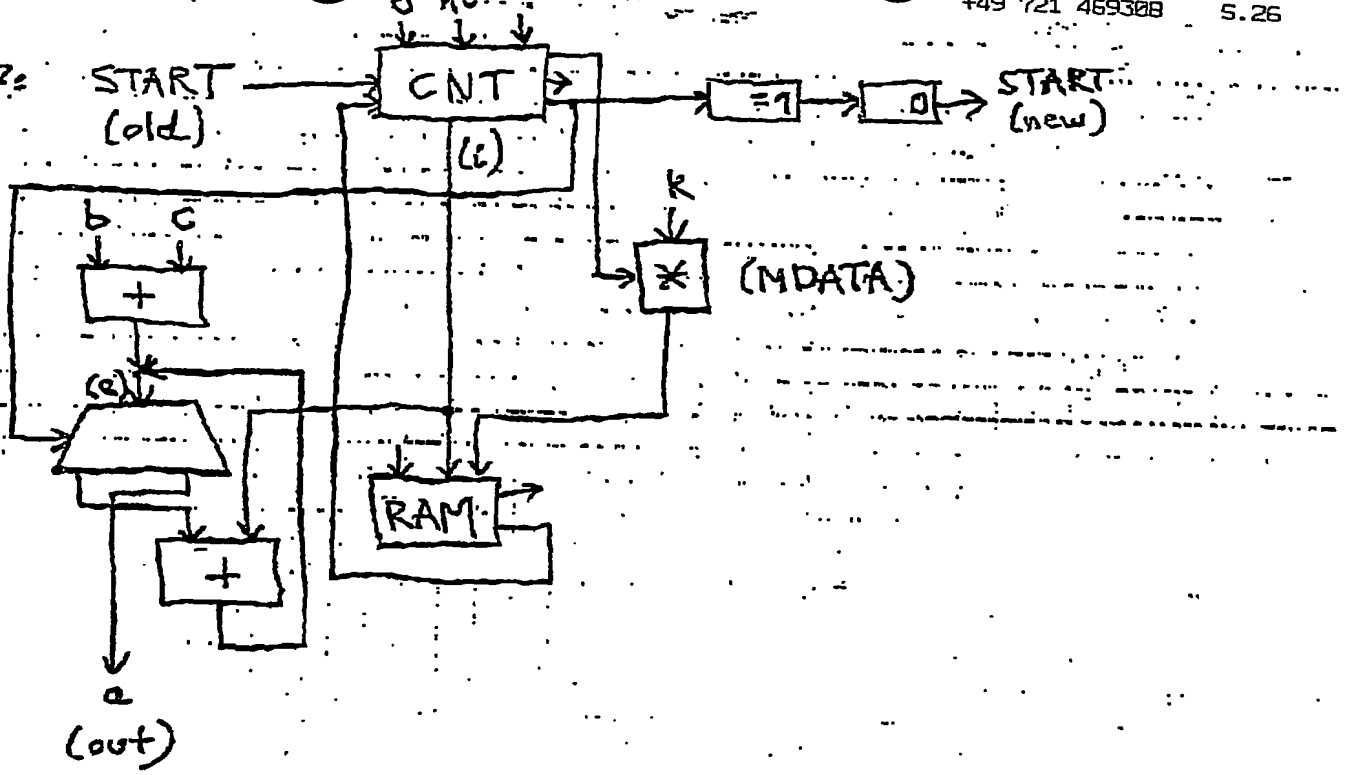


Fig 8:
General Conditional
Statement Template

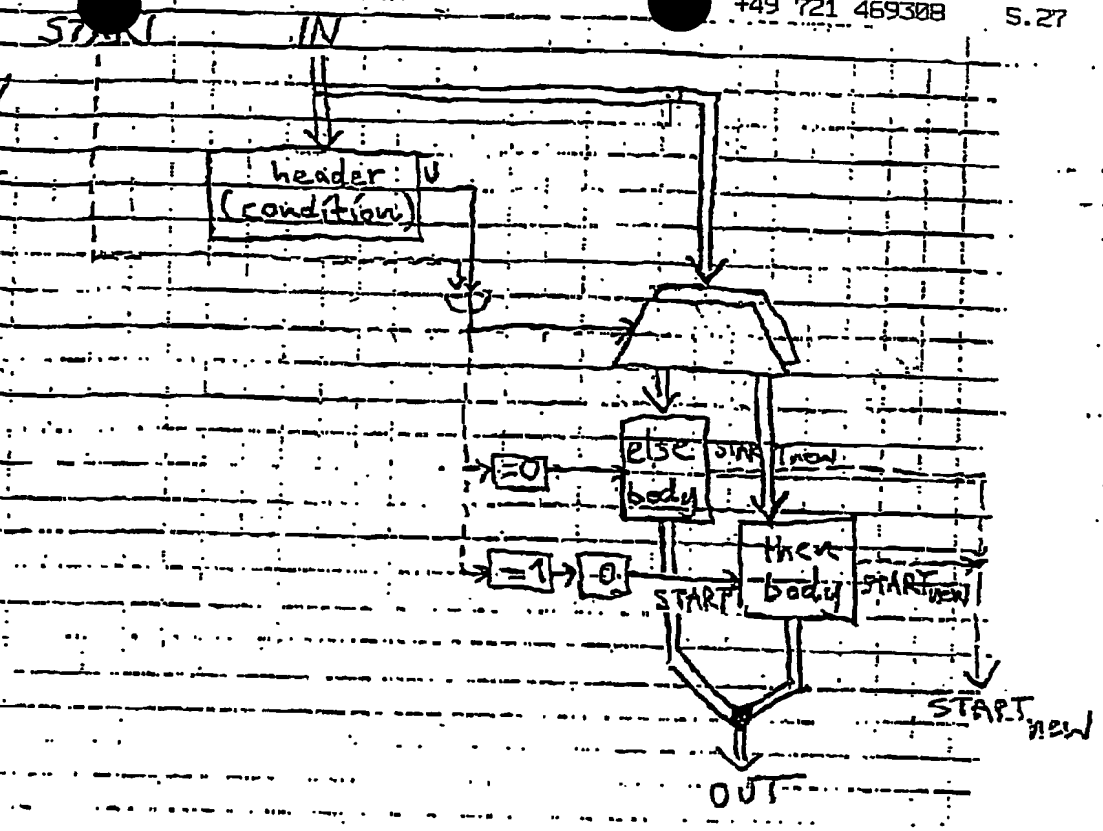
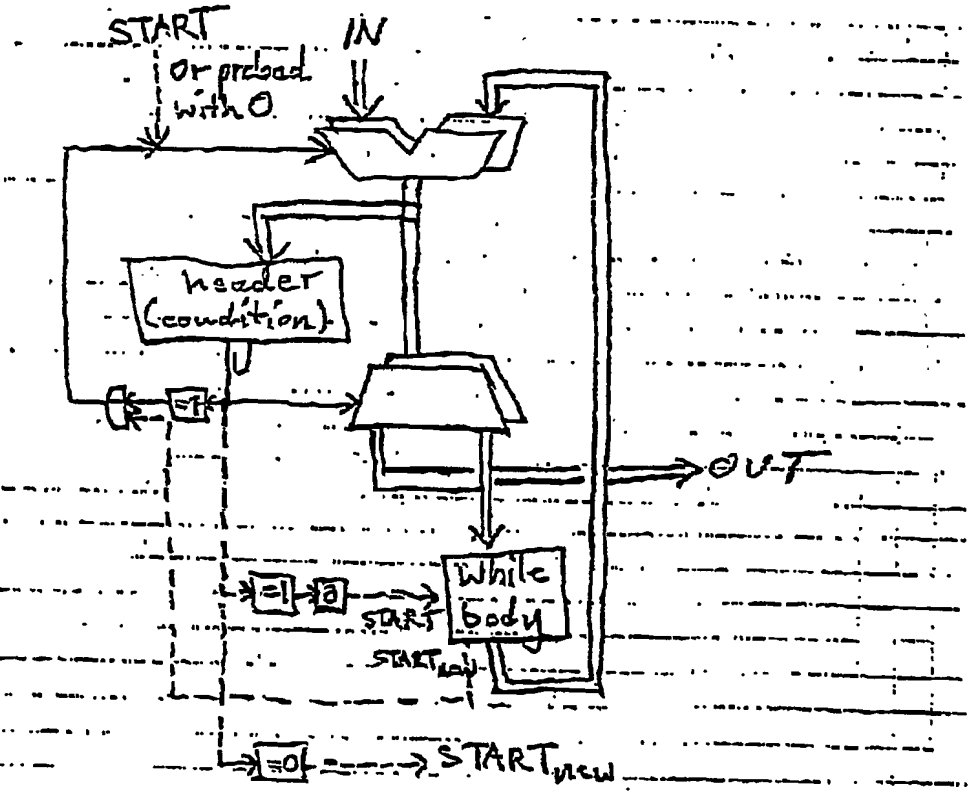


Fig. 9:
While Loop Template



g. 10=

or Loop Template

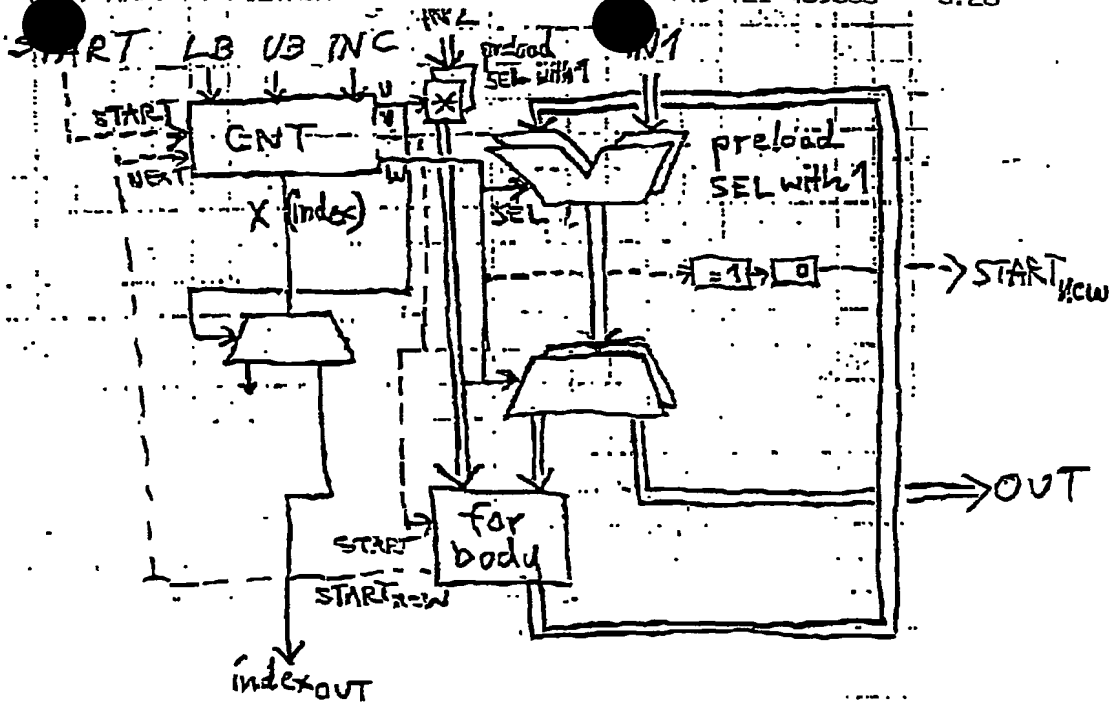
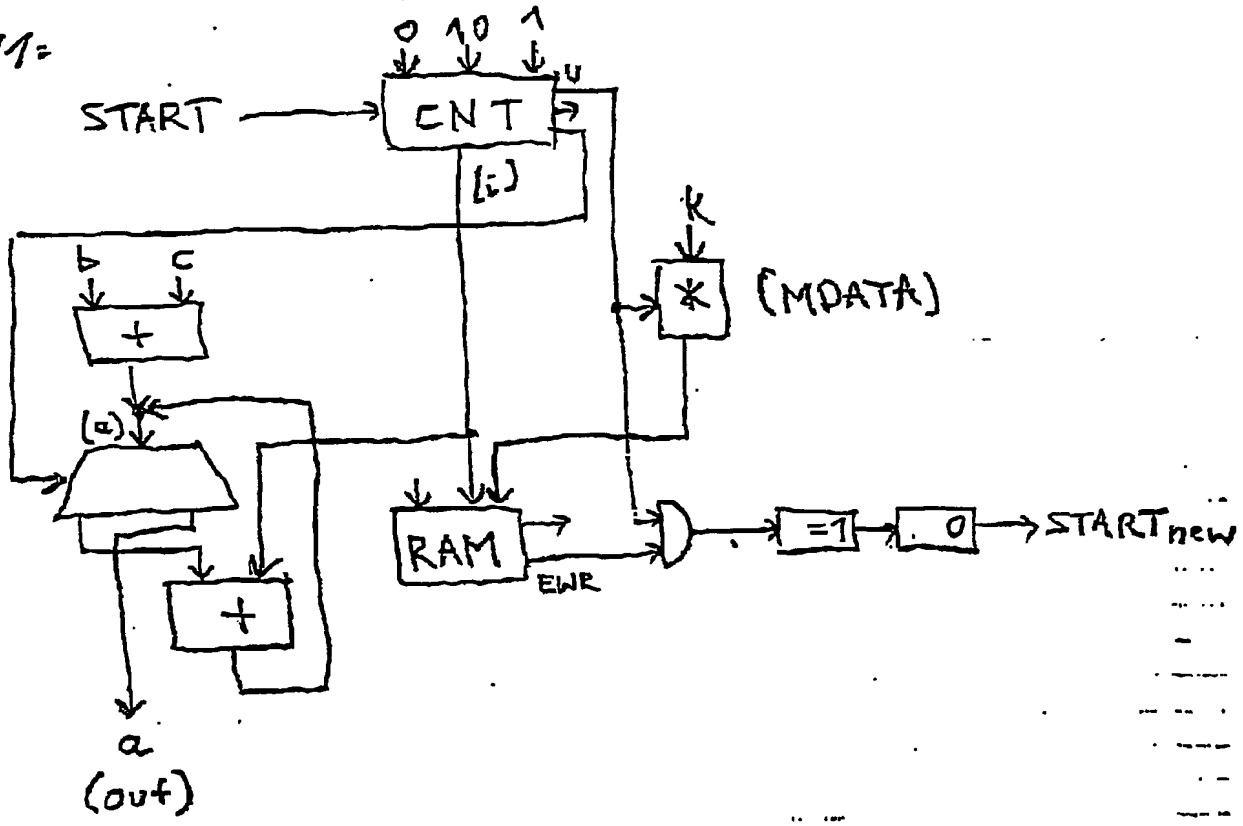


Fig. 11=



PACT27a IEP

6/9

Fig. 12: START (old)

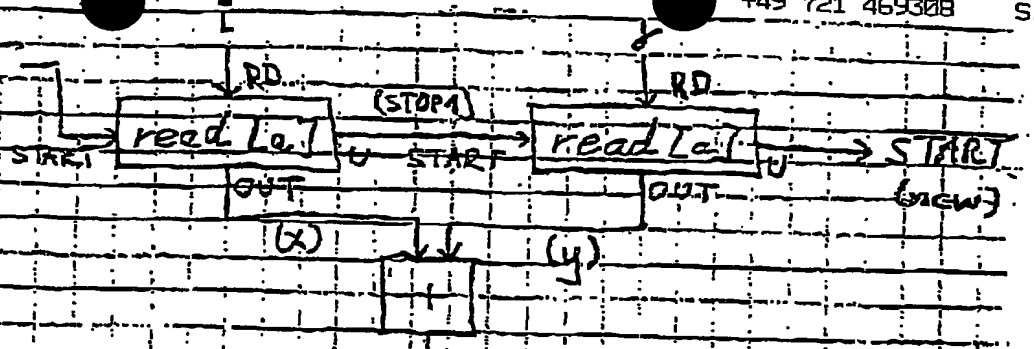


Fig. 13: START$_a$

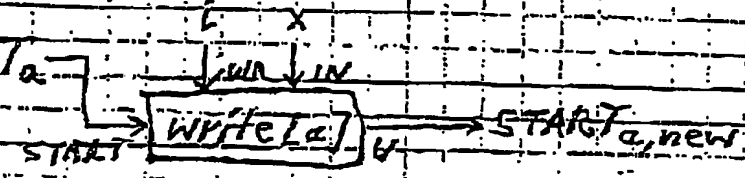
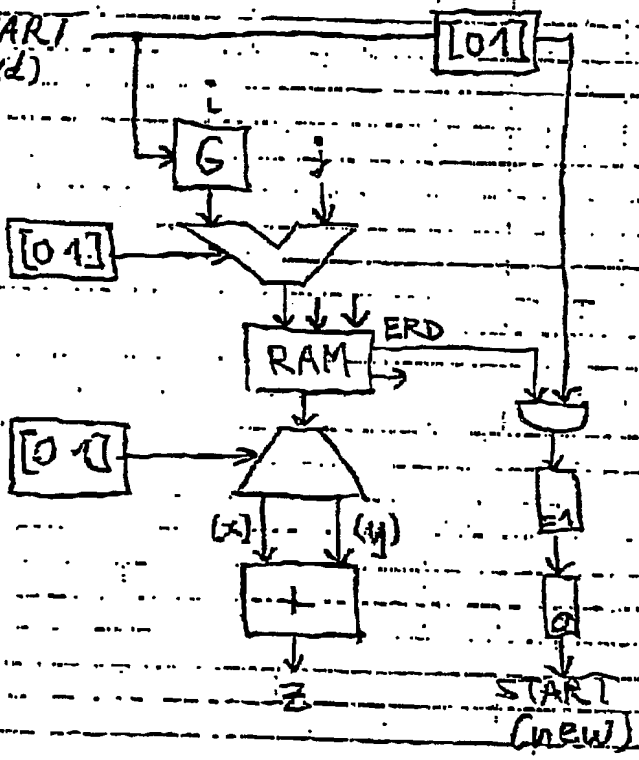


Fig. 14: START (old)



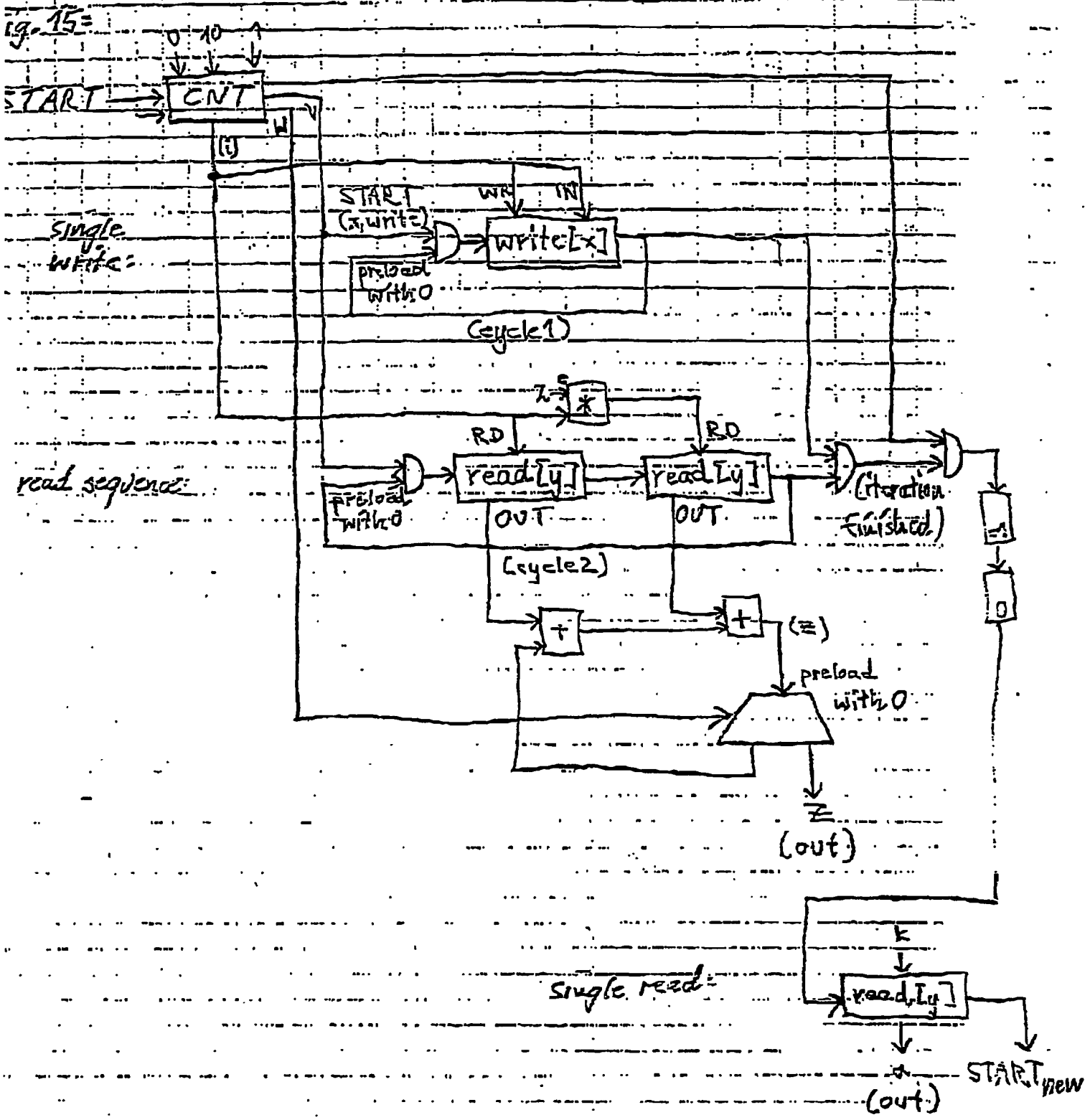
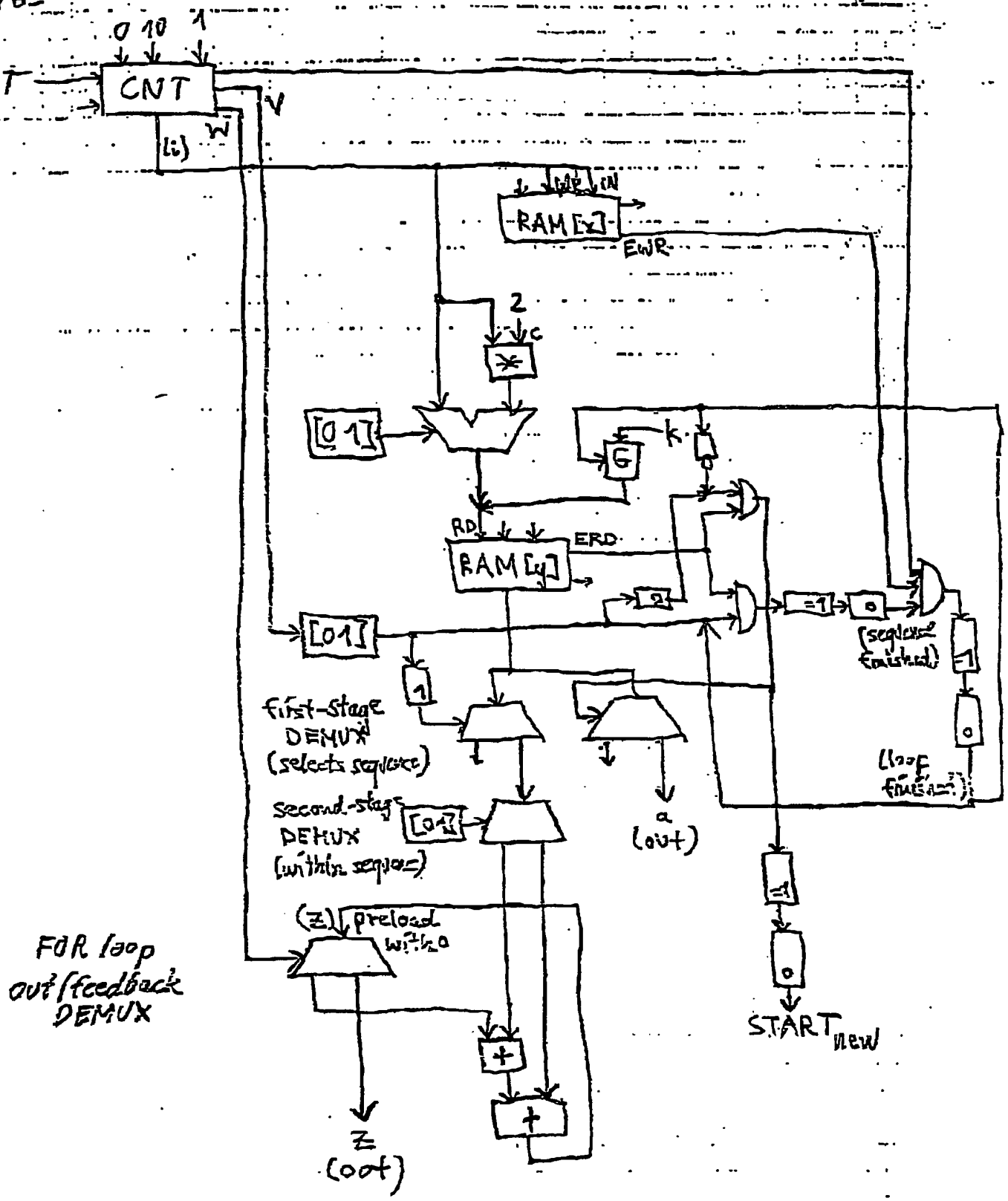


fig. 15:
START
CNT
0
10
1
N
M
IJ
single write:
START (x,write)
WR
IN
write[x]
preload with 0
(cycle1)
read sequence:
RD
RD
read[y]
read[y]
preload with 0
OUT
OUT
(iteration finished)
(cycle2)
+
+
(=)
preload with 0
Z
(out)
=1
0
K
single read:
read[y]
(out)
START new

Fig. 16


START
CNT
0
10
1
V
W
RAM [z]
EWR
2
c
[0 1]
G
k
RD
ERD
RAM [y]
[0 1]
=1
(sequence finished)
first-stage DEMUX (selects sequence)
second-stage DEMUX (within sequence)
[0 1]
a (out)
(z) preload with 0
FOR loop out (feedback DEMUX
START new
z (out)